An Experience with Two Symbolic Execution-Based Approaches to Formal Verification of Ada Tasking Programs

Laura K. Dillon Richard A. Kemmerer Linda J. Harrison
Computer Science Department
University of California, Santa Barbara, CA 93106

Abstract
There have been several efforts to use symbolic execution to test and analyze concurrent programs. Recently proof systems have also emerged for concurrent programs and for the Ada language in particular. This paper reports on an experience with developing two different approaches, which use symbolic execution, to prove partial correctness and general safety properties of Ada programs. One approach is based upon interleaving the task components while the other is based upon verifying the tasks in isolation and then performing cooperation proofs. Both approaches extend past efforts by incorporating tasking proof rules into the symbolic executor allowing Ada programs with tasking to be formally verified.

The limitations of each approach are presented, along with each approach's advantages and disadvantages. In particular, the difficulty of dealing with communication statements in a loop structure are addressed in detail.

Index Terms - Symbolic execution, Formal specification and verification, Program validation, Testing, Software tools, Programming languages.

1 Introduction
Symbolic execution is a method, which has been used successfully in the past to test and prove sequential programs [4,9,13,14]. With this approach algebraic symbols are used to represent the input values rather than using numeric or character inputs. These symbols are then manipulated by the program. By placing restrictions on the values that each symbol may represent the symbolic input represents a class of input values rather than a single value. That is, by properly restricting the input values each symbolic execution of a program can correspond to executing the program on a particular subdomain. When execution is completed the symbolic values obtained by the variables are analyzed to determine if they meet the programmer’s expectations. The tool that performs the symbolic execution is called a symbolic executor.

A symbolic executor can also be used to generate the necessary verification conditions that need to be proved to verify that a formally specified program is consistent with its entry and exit specifications. The type of correctness being verified is partial correctness as originally defined by Floyd [7]. That is, if the entry assertion is true when the program is invoked and the program terminates, then the exit assertion will be true when the program halts. A path condition is used to summarize the relations which hold between the values of variables as the program executes. In general, when an assertion is reached, the path condition should be strong enough to imply the correctness of the assertion.

Recently there has been some work to extend symbolic execution to test concurrent Ada programs [15,22]. These extensions, however, have not included developing proof rules so that the programs can be formally verified using symbolic execution. At the same time Hoare style axiomatic proof rules have been developed for subsets of the Ada language that include tasking [2,8]. In this paper two approaches to symbolic execution, which incorporate tasking proof rules, are presented and compared. One approach is based on interleaving the execution of component tasks to simulate their concurrent execution. The other is based on proof systems that allow independent verification of component tasks, followed by a test for “cooperation”.

The next section presents the tasking subset of the Ada language that is considered in this paper. Section 3 introduces the interleaving approach, and presents a simple example without loops followed by an example with loops. Problems encountered when the loop contains communication statements are discussed and experiences with several approaches to solving this problem are analyzed. Section 4 introduces the isolation approach and the results of applying it to the looping example. In section 5 the two approaches are compared, and section 6 relates the work reported here to previous work with concurrent programs. Finally, some conclusions and future work are presented.

In the remainder of this paper the reader is assumed to have a working knowledge of both the Ada language and the symbolic execution approach to formally verifying sequential programs. The reader is directed to [9] for an excellent introduction to the latter.

TH0225-3/88/0000/0114$01.00 © 1988 IEEE 114
2 The Ada Environment

A tasking subset of the Ada language is considered in this paper. The subset is defined by the following assumptions.

- There are no shared variables.
- There is no aliasing.
- Assignments have no side effects.
- There is no exception handling.
- The abort, delay, and terminate statements are not allowed.
- There are no task attributes.
- There is no dynamic allocation of tasks.
- All tasks are declared in the top level procedure.
- There is no nested rendezvous.

Although Ada permits inter-task communication both through the use of shared variables, and the rendezvous mechanism, shared variables are disallowed in this subset. Introducing shared variables complicates both the proof rules, and the granularity required of concurrent execution unit interleaving. Both of the approaches to symbolic execution considered in this paper can be adapted to allow nested rendezvous [6,11]. However, for simplicity, nested rendezvous is not considered here.

To use symbolic execution to verify the consistency of a program and its specification it is necessary to extend the Ada language to include entry, exit and loop assertions. Each concurrent execution unit may have an entry assertion and must have an exit assertion. If there is no entry assertion for a unit, it is assumed to be true. Every accept body also has an entry and exit assertion. If there is no accept body the entry and exit assertion are assumed to be true. As in sequential programs, all loops must have a loop assertion, and all procedures have entry and exit assertions. The assertions are first order predicate logic formulae similar to those used in the Anna language [18].

3 The Interleaving Approach

In the interleaving approach to symbolic execution parallelism is modeled in the standard way by arbitrary interleavings. Only one task is active at a time, and there may be many different interleavings of tasks from a single program state. The start of each symbolic execution path corresponds to choosing a different task to be the next to execute. Taken collectively all of the paths starting at the initial state represent all possible execution paths through the program. The non-deterministic nature of concurrent programs makes it impossible to know which execution path the program will actually execute; therefore, all paths must be considered. That is, the interleaving strategy must guarantee that all possible execution paths through a program are considered, and no path should be considered more than once. The details of the interleaving approach discussed in this paper are presented in [11]. A mechanism to generate all paths is also presented there. Because shared variables are not allowed the interleaving points are at blocked entry call and accept statements, after a rendezvous, and at task completion.

The data structures that are used in the interleaving approach presented in this paper are as follows. Each task has a symbol table, denoted SYMi[], which stores the symbolic values for all variables in the task. If e represents an expression, then SYMi[e] denotes the value of expression e, where the variables within the expression are evaluated by looking up their values in the symbol table for task i.

There is also a path condition, PCi, for each task. The path condition is an expression that records constraints known about the relations between the values of variables in the symbol table. To keep track of the tasks that block at entry call or accept statements it is also necessary to have a wait set and a calling list set for each task. The wait set for task i, WSi, contains the names of all entries on which this task is currently blocked waiting for an entry call. A task’s calling list set contains one calling list for each entry in the task. The calling list for entry j of task i is denoted, CLij. Each calling list contains the names of tasks which have called this task and are currently blocked waiting for a rendezvous. Finally, each task has its own location counter, LCi, which indicates what statement is to be executed next.

The state for a task i is then:

\[ TS_i = SYMi[], PC_i, WS_i, CL_{1i}, \ldots, CL_{Mi}, LC_i \]

where M is the number of entries in task i.

The currently active task is denoted CT. All other tasks are either able to run, blocked waiting for a rendezvous, or have completed execution. The ready set, RS, contains the names of all of the tasks that are able to run. The completed set, CS, contains the names of all tasks that have completed their execution.

The program state, which is the state of all the tasks taken together, is:

\[ PS = TS_1, \ldots, TS_N, RS, CS, CT \]

where N is the total number of tasks.

When symbolically executing a program the execution of a program statement alters the program state according to the symbolic execution rule for that type of statement. Therefore, it is necessary to have a symbolic execution rule for every statement type in the language. As an example, a rule for the symbolic execution of an assignment statement, \[ x := e \], in task i is given below.

Assignment Statement Rule:

\[ SYMi[e] := SYMi[\text{SYMi}[e] \rightarrow x] \]

where SYMi[v \rightarrow x] denotes changing the value of x to v in the symbol table for task i. Thus, the expression e is evaluated by replacing the variables in e with their current values, as indicated in the symbol table for task i. The resultant value becomes the value associated with variable.
\( z \) in \( SYM_i \) and all other variables in \( SYM_i \) retain their old values. Only the symbol table for task \( i \) is changed.

Proof rules for program statements unique to the Ada tasking constructs are presented in [11]. In particular, rules for task entry and exit, entry call, accept, and select statements are given. Briefly, whenever a rendezvous occurs the path conditions of the communicating tasks are used to prove the entry assertion for the accept body. Next fresh symbolic values are assigned to the actual out and in out parameters and they are used to evaluate the exit assertion for the accept body, which is conjoined to the path conditions of the communicating tasks.

### 3.1 A Simple Example

The example below, which is adapted from [8], illustrates the use of the interleaving approach to symbolic execution.

**Example 1**

```plaintext
1 task body T is
2 begin
3 T1.A(1);
4 end T;

1 task body T2 is
2 begin
3 accept A(U:in integer) do
4 X := U;
5 end T;
```

Figure 1 shows a partial symbolic execution tree for this example. It details the first few steps of one of the execution paths through the program. The ellipses represent program states at interleaving points. The label inside the ellipse provides a convenient notation for identifying the state. The numbers represent the location counter (LC) program states at interleaving points. The label inside the ellipse provides a convenient notation for identifying the execution paths through the program. The ellipses represent path conditions through the program. The ellipses represent path conditions through the program. The ellipses represent path conditions through the program. The ellipses represent path conditions through the program.

### 3.2 A Looping Example

**Loop invariants** are introduced when verifying sequential programs to reduce potentially infinite symbolic execution trees to finite ones. The same approach can be used with tasking programs in which the loop statements do not contain communication statements. The approach is based on finite induction, where the induction is on the number of iterations of the loop. The idea is that the loop invariant characterizes everything that needs to be known about a task's computation when it reaches that point. It is necessary to show that the loop invariant holds the first time it is encountered (the basis case). Then starting with fresh symbolic values for all of the variables of the current task and the evaluated loop invariant as the new value for the task's path condition, the loop is executed and the loop invariant must be shown to hold when it is encountered again (the inductive step). When the loop invariant is reached the second time and the necessary VC is generated the execution of this branch of the tree is complete. Note that further execution of the branch is unnecessary,
as it would continue from a state that is virtually identi-
cal to the ancestor state generated after the loop invariant
was first encountered.

This approach fails when applied to a loop containing
a communication statement. The reason it fails is that
the loop invariant characterizes everything that is known
about the task containing the loop, but not about other
tasks. The program state \( (PS) \) for a tasking program,
however, is determined by the states of all tasks. The com-
munication statements within the loop body allow other
tasks to execute after the loop invariant is first encoun-
tered and before it is next encountered. Therefore, when
the loop invariant is reached the second time it is not nec-
essarily the case that, if execution of this branch were to
continue, it would continue from a state that has already
been encountered.

The pedagogical example below, which has a pair of
tasks with loops containing communication statements, il-
lustrates this problem.

**Example 2**

{Entry: \( N \geq 0 \)}
1 task body \( T_1 \) is
2 \( I: \text{integer} := 0; \)
3 begin
4 while \( I < N \) loop
5 accept \( E; \)
6 \( I := I + 1; \)
7 end loop;
8 end \( T_1; \)
--- {Assert: \( I < N \)}

{Entry: \( N \geq 0 \)}
1 task body \( T_2 \) is
2 \( J: \text{integer} := 0; \)
3 begin
4 while \( J < N \) loop
5 \( J := J + 1; \)
6 end loop;
7 end \( T_2; \)
--- {Assert: \( J < N \)}

Figure 2 shows a partial symbolic execution tree for the
example, obtained using the approach described above.
Here, circles are used to indicate branch points in the ex-
ecuting task, and ovals are used as in the previous example.
To illustrate the loop invariant problem described above,
consider state \([4,6]\) in this figure. This is the second time
that \( T_1 \)'s loop invariant has been encountered (where the
first time was state \([4,3]\)). However, these are different
program states because \( T_2 \) is at a different point in its
computation. It is therefore premature to stop execution
of \( T_1 \) in state \([4,6]\). This problem is addressed by chang-
ing the proof rules. Instead of terminating the task's exe-
cution when the loop invariant is encountered a subsequent
time, the loop is iterated again. Unfortunately, this may
result in an infinite symbolic execution tree. The issue of
producing finite trees is discussed below.

Example 2 reveals a second problem; *infeasible paths*
(i.e., paths that can never be followed during actual ex-
ecution of the program) may be generated. In figure 2,
for instance, the paths leading to \([8,5]\) and \([5,8]\) are in-
feasible. The loop invariant must be strong enough to
eliminate infeasible branches. Otherwise, spurious errors,
such as these apparent deadlocks (indicated by the empty
ready sets while some tasks have not completed), will be
reported.

Closer examination of the program provides insight into
this problem. The specification does not capture the cor-
respondence between the exit conditions of the loop state-
ments in \( T_1 \) and \( T_2 \). One way to capture this corre-
spondence is to strengthen the loop invariant so that it
expresses the relationship between the values of \( I \) and \( J \).
However, the relationship between \( I \) and \( J \) is dependent
on the value of the location counters in \( T_1 \) and \( T_2 \). For in-
stance, if both tasks are at the head of the loop statement,
then \( I = J \). If, however, \( T_1 \) is executed after a rendezvous
on \( T_1.E \) and reaches its statement 4 before \( T_2 \) executes
its statement 6, then \( I = J + 1 \). Thus, assertions in a task
are allowed to reference variables and location counters
of other tasks. Consider example 2. The loop invariants
are strengthened to express the relationship between the
values of \( I \) and \( J \) as follows.

In \( T_1 \):
- \( I \leq N \) and \( \text{LC}_2 \neq 6 \Rightarrow I = J + 1 \)
- \( \text{LC}_2 = 6 \Rightarrow I = J + 1 \)

In \( T_2 \):
- \( J \leq N \) and \( \text{LC}_1 \neq 6 \Rightarrow J = I + 1 \)
- \( \text{LC}_1 = 6 \Rightarrow J = I + 1 \)

Now when a loop invariant is encountered during execu-
tion, the PCs for both T1 and T2 may be needed to establish the validity of the invariant. Hence, the conjunction of their PCs provide the antecedents for the VCs generated on encountering the loop invariant.

There remains the issue of producing finite trees. One way to guarantee termination of symbolic execution is to assign fresh values to all variables in all tasks when a loop invariant is encountered and to use the invariant to obtain a new global PC. This essentially treats the loop invariant as expressing everything that needs to be known about the state of all tasks for any execution in which the current task reaches the point associated with the invariant. It was felt, however, that this approach places unreasonable demands on the specifier to determine the appropriate loop invariant. The problem of inventing strong enough loop invariants would be too hard for the approach to be of practical use. In particular, the loop invariants given above would not suffice. The decision was made, therefore, to assign fresh values to those variables in the current task only. In this way, the loop invariant need only express what must be known about the values of the current task’s variables and their relationship to the values of the other tasks’ variables.

Figure 3 details part of the symbolic execution tree produced from the program in example 2 using this approach and the strengthened loop invariants. Note that the infeasibility of the paths leading to the states [8, 5] and [5, 8] is now evident, as the PCs for T1 and T2 are contradictory. Pruning can be used in this example to produce a finite execution tree. Further research is required to determine if this approach guarantees finite execution trees in general.

Young and Taylor discuss the problem of producing finite symbolic execution trees in [22]. They outline a solution that relies on loop invariants and assertions following rendezvous. It is not clear from their description if assertions in a task can refer to variables in other tasks nor if strong enough loop assertions can be obtained to make it possible to detect infeasible paths.

4 The Isolation Approach

The isolation approach overcomes many of the shortcomings of the interleaving approach. In general, however, it requires a stronger and less intuitive specification. Based on the proof systems of [1, 2, 8], the approach allows independent verification of component processes in local proofs, which are later checked for cooperation. The basic idea is as follows. In order to view a task in isolation, assumptions must be made about the environment whenever communication takes place. These assumptions are expressed in assertions associated with communication statements (i.e., entry call statements and accept statements). Such assertions are assumed valid for the local proof of a task. The cooperation proof shows that the local proofs cooperate in validating these assertions, so that the proof is well-founded.

In the isolation approach, the only variables that may appear free in the assertions for a task are the task’s local variables and the program constants. This limits the possibility of interference between the local proofs of tasks to the points at which they communicate. However, it may not be possible to express certain information required for the cooperation proof, such as a task’s locus of control or the source of a message that is received in rendezvous, using assertions over a task’s variables. The specification of a concurrent program is therefore extended to allow the introduction of auxiliary variables (AVs), a global invariant (GI), and bracketed sections. The AVs for a task encode information pertaining to the task that is required for the cooperation proof. Relationships between the variables of different tasks are expressed in the GI.

A bracketed section contains a single communication statement, which is (optionally) preceded and/or followed by a sequence of assignment statements. All updates to the GI variables and all communication must be enclosed in (non-overlapping) bracketed sections. Thus, bracketed sections delimit regions in which interference can occur. Cooperation must be established for each pair of matching bracketed sections. The GI is required to hold except within bracketed sections. In the following example, the program from example 2 is respecified for use with the isolation approach. Bracketed sections (delimited by angle

\[ \begin{align*}
&I: 0; N: \delta \\
&VC: \delta \geq 0 \Rightarrow \phi = 0 \leq \delta \\
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ T_1 \}
\end{align*} \]

Figure 3: Symbolic execution tree for the program in example 2 using the strengthened loop invariants

\[ \begin{align*}
&PC_2: a_1 = \delta \leq \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 < \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 < \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_1: a_1 = \beta_1 = \delta \\
&RS_1: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 = \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 = \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&I: 0; N: \delta \\
&VC: \delta \geq 0 \Rightarrow \phi = 0 \leq \delta \\
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ T_1 \}
\end{align*} \]

\[ \begin{align*}
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 < \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 < \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 = \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 = \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&I: 0; N: \delta \\
&VC: \delta \geq 0 \Rightarrow \phi = 0 \leq \delta \\
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ T_1 \}
\end{align*} \]

\[ \begin{align*}
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 < \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 < \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 = \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 = \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&I: 0; N: \delta \\
&VC: \delta \geq 0 \Rightarrow \phi = 0 \leq \delta \\
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ T_1 \}
\end{align*} \]

\[ \begin{align*}
&PC_1: a_1 = \delta \leq \delta \\
&RS_1: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 < \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 < \delta \\
&RS_3: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_2: a_1 = \beta_1 = \delta \\
&RS_2: \{ \}
\end{align*} \]

\[ \begin{align*}
&PC_3: a_1 = \beta_1 = \delta \\
&RS_3: \{ \}
\end{align*} \]
brackets) and a GI are introduced. No AVs are required for this simple program.

Example 3

```
1 begin
2 I:=I+1;
3 end loop;
4 while I < N loop
5 accept E;
6 J:=J+1;
7 end loop;
8 end T1;
- - {Exit: J = N}
- - {Global invariant: I = J}
- - {Entry: N >= 0}
```

The symbolic execution units (i.e., the top level procedure, the task bodies and the accept statement bodies) in a program are executed independently of one another in the isolation approach. Symbolic execution of a unit produces a set of task states, each consisting of an evaluation map, a path condition, and a location counter. The evaluation map, path condition and location counter in a task state for each matching accept statement represent the computation in which the accept statement is symbolically executed in a given task state. The rules for executing sequential programming constructs are the standard ones. Symbolic execution of a unit proceeds under the assumption that all other units satisfy their specifications. Thus, an entry call statement is viewed as a call to a "procedure" whose specification is given by the entry and exit assertions associated with it. The local proofs of the tasks in example 3. The first ensures that the GI holds after elaboration of the declarative parts of T1 and T2. It is obtained in the obvious fashion from T1's and T2's task states 4.2. In

The cooperation proof is necessary to demonstrate that the assumptions made in the local proofs of the units are valid and that the tasks cooperate in establishing and maintaining the GI. Two VCs are generated for proving cooperation between the local proofs of the tasks in example 3. The first ensures that the GI holds after elaboration of the declarative parts of T1 and T2. It is obtained in the obvious fashion from T1's and T2's task states 3. The second ensures that the GI is preserved by parallel execution of matching bracketed sections. Assuming that the GI is valid at the beginning of a task state, the generated VC is, \(\alpha_1 < \delta \land \beta_1 < \delta \land \alpha_1 = \beta_1 \Rightarrow \alpha_1 + 1 = \beta_1 + 1\). The antecedent of this VC is derived from T1's and T2's task states 5 by asserting the GI, while the consequent is the GI, evaluated in T1's and T2's task states 4.2. In general, an accept statement may have an entry assertion associated with it. The local proofs of tasks with matching bracketed sections are shown to cooperate in establishing the entry assertion for the embedded accept statement by similar VCs.

The major hurdle to successful use of the isolation approach for verification is in producing an appropriate spec-
ification for a program. The information required for the cooperation proof is seldom evident. This point can be appreciated by considering the program in example 1, which can be verified (without introducing AVs or a GI) using the interleaving approach. When the task T is viewed in isolation, there is no way of knowing that the call to T1.A cannot result in execution of the second accept statement in the body of T1. When the isolation approach is used, therefore, AVs and a GI are required to express relationships between the progress of the tasks. One possible specification, for example, introduces a boolean-valued AV for each task, say F, F1 and F2 for the tasks T, T1 and T2, respectively. The F, F1, and F2 are initially false, and are set true in each of the corresponding task's first bracketed section (signaling that the task has engaged in its first communication event). The GI defined by 

\[(F \text{ iff } F1) \text{ and } (F2 \text{ implies } F1)\]

contradicts the path conditions associated with those pairs of (syntactically) matching entry call and accept statements that do not match semantically. Thus, the antecedents of the VC's that are generated to demonstrate cooperation for such pairs are, false, making the VC's trivial theorems.

5 Comparison of the Approaches

The main advantage of the interleaving approach is that the states of all tasks are known in any symbolic execution state. Therefore, it is not necessary to introduce a global invariant, auxiliary variables or bracketed sections. Producing a specification which assures cooperation is one of the major difficulties of the isolation approach. In particular, the global invariant must trivialize the VC's generated for all non-semantic matches between entry calls and accepts. In the interleaving approach this is only an issue if the communication statements appear in the body of a loop.

The primary disadvantage of using the interleaving approach is the number of program states that are necessary to have a complete symbolic execution of a program. Reducing the granularity of the interleaving of task executions by partitioning tasks into sequential segments and interleaving the segments, as illustrated above, helps to reduce this problem. Any program state where it is necessary to pick the next task to execute is the head of m paths where m is the number of task names in the ready set for that state. At most one communication statement can appear in each sequential segment. The number of states, therefore, still grows exponentially with the number of communication statements in a task. It is important, moreover, to formally justify the particular choice of interleaving so that potential behaviors are not overlooked. Further research is required to address this issue.

The isolation approach also suffers from combinatorial problems, but to a lesser degree. Here, the explosion is in the number of VC's generated for the cooperation proof, which is polynomial in the number of matching communication statements and the maximum number of task states whose location counters designate the same communication statement. Examples show that, in a loosely coupled system, this represents a dramatic reduction over the number of interleavings of sequential code segments determined by all communication statements.

The difference in the combinatorics of the two approaches is appreciated by considering task families. The complexity in the isolation approach is tied to the number of task types, not the actual number of tasks. On the other hand, the complexity of the interleaved approach increases with each additional task instance.

The isolation approach is more compositional than the interleaving approach. Tasks cannot be verified independently using the interleaving approach. Any modification to a task body requires re-execution of the entire program. Using the isolation approach, the local proof of a task can be made as soon as the task body and specifications for the accept statements that it calls are present. Furthermore, a modification to a task body does not affect the validity of the local proofs of other tasks unless the modified task's specification is also changed. The VC's in the cooperation proof that involve the modified task may be affected, but no other VC's need be re-generated.

In [5] it is shown how symbolic execution trees and VC's can be generated in an incremental fashion from a sequence of progressively "stronger" specifications of a program using the isolation approach. This requires retaining certain key task states generated for the local proofs, along with the antecedents of the VC's, from which changes in task states and new VC's are generated. The same technique cannot be used with the interleaving approach because of the large number of program states that would have to be retained. This suggests that the isolation approach will facilitate a more experimental approach to verification. It should be noted that in both approaches the actual process of carrying out the symbolic execution and the corresponding proof of the generated VC's is not complex. It is an error prone job for humans, but is perfectly suited for automation.

6 Related Work

Interleaving approaches to symbolic execution of concurrent Ada programs are discussed in [15,22]. As mentioned above, these works are primarily concerned with symbolic testing, rather than verification, and no proof rules are presented. The interleaving approach in [3] is used for verification, but for a low-level concurrent programming language.

Numerous axiomatic proof systems for concurrent programs have been explored [1,2,8,12,16,17,19,20,21]. Apt [1] introduced the idea of cooperation between proofs to
deal with communication and synchronization in CSP programs. The idea of cooperation was subsequently applied, in both [2] and [8], to model communication by rendezvous in Ada. The isolation approach builds on these later two proof systems.

AVs and a G1 figure prominently in [1,2,8,21]. Hoare's "channel variables" are essentially AVs, while his assertions are invariants [12]. Lamport's GHL [17] uses invariants exclusively, which are made conditional on control predicates. The use of control predicates in assertions is equivalent to referencing the LCs of tasks. The LCs can be viewed as special-purpose AVs, which are automatically updated.

### 7 Conclusions and Future Work

The use of two different symbolic execution approaches that deal with a significant tasking subset of Ada were presented. The approaches are interleaving and isolation with cooperation proofs. Two examples were analyzed and possible solutions to the problems raised by communication statements in a loop structure were discussed.

The comparative strengths and weaknesses of the two approaches suggests that the isolation approach may be better suited for formal verification, while the interleaving approach may be useful as a basis for symbolic testing and analysis. The Reliable Software Group at UCSB has implemented a symbolic executor, called UNISEX, that can be used for formally specifying, testing, and verifying Pascal programs [14]. Experience with verifying sequential programs using UNISEX indicates that programs are typically verified in an experimental fashion. The "obvious" assertions that express one's understanding of a program are typically too weak. However, by examining the VCs generated from an initial specification, the specifier gains a deeper understanding of the program and identifies "missing" assumptions to add to the current assertions. The isolation approach supports precisely this type of experimental verification paradigm [5]. In fact, it encourages an incremental approach to verification of safety properties. It is not necessary to produce a single monolithic specification that expresses the information required to prove all desired safety properties of a program. Rather, verification can begin with a specification designed to express the information required for proving a single safety property. The specification can then be strengthened, as necessary, to gradually verify each desired safety property in turn. Clearly, the verification of a program is also facilitated by the compositional nature of the isolation approach.

When designing the proof rules for the two approaches the problem of state explosion and the need to develop sophisticated pruning techniques to deal with the combinatoric disaster in the interleaving approach was anticipated. In contrast, the infeasible path generation problem that occurs when loops contain communication statements was not anticipated. In fact, the work primarily had been concerned with representing the central issues in tasking—concurrent execution of tasks and communication by rendezvous. As a result symbolic execution rules for loop structures were not even included in [11]. Currently the Reliable Software Group is designing a UNISEX-like symbolic executor based on the interleaving approach.

The question of what variables need to be assigned fresh symbolic values when starting from a loop invariant is still open. The worst case would require assigning fresh values to all variables in all tasks, as described above. This would place an unreasonable burden on the specifier who had to discover the appropriate loop invariants. The approach used in the example was to assign fresh values only to the executing task's variables. Although this worked for example 2, it may not be sufficient for all other programs. This is an area that deserves further research.

Another area that requires more work is the choice of interleaving points. The interleaving approach presented in this paper chose a task to execute after elaboration of the declarative parts of all tasks, at a blocked entry or accept, after a rendezvous, and at task completion. Since there are no shared variables, intuitively this seems adequate to model all possible interleavings that could actually occur. However, if the interleaving approach is to be pursued it will be necessary to formally justify the particular choice of interleaving points.

As mentioned above the major drawback of the interleaving approach is the large number of states that may be produced when using it. To address this concern, methods of pruning duplicate paths are being pursued. Two of the areas being investigated are methods for hashing program states to allow efficient comparisons, and code interchange techniques used in optimizing compilers.

The size of the program state space is less problematic for symbolic testing than verification. An interleaving symbolic executor can be used to interactively analyze specific program paths, and automatically generate VCs to check assertions as they are encountered. The advantage to using the interleaving approach, rather than the isolation approach, for symbolic testing is that specific paths can be explored to determine the resulting program state, which is given in terms of the values of the actual program variables. The notion of program path does not exist in the isolation approach. A "program state" for specific configurations could be constructed from the local proofs of tasks. These states, however, would rely on the values of auxiliary variables to express relationships between the actual program variables, and so would be harder to interpret.

Finally, the need to develop symbolic execution rules for a larger subset of Ada is evident. Obvious candidates are conditional entry call statements, delay statements, and exception handling [10]. Research is required to identify restrictions that can be placed on the use of shared variables so that the approaches can be extended to allow
communication by sharing data as well as by rendezvous. Extensions that embed timing information in the specification are also being explored. Some of these extensions are obvious and only require time to do, while others pose challenging research questions.

References


